Dzisiejszy wykład

- Hierarchie klas i rzutowanie
- Informacja o typach w czasie wykonania (RTTI)
- Wskaźniki do składowych
- Operatory *new* i *delete*
- Obiekty tymczasowe

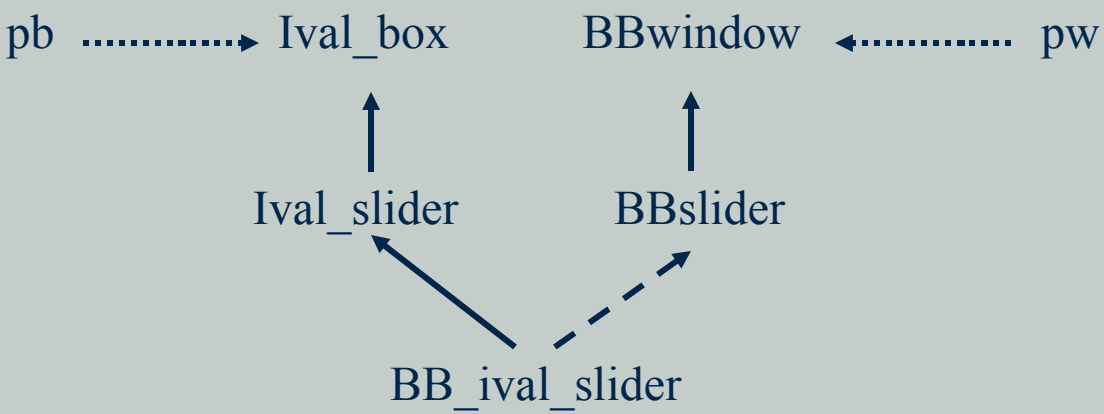
Rzutowanie

- Sensownym użyciem klasy *Ival_box* jest przekazanie obiektów tego typu do systemu kontrolującego ekran i spowodowaniu, by system przekazywał obiekty z powrotem do programu użytkowego, gdy coś się zacznie dziać
- Systemowy interfejs użytkownika nie zna naszej klasy, jest wyspecyfikowany w kategoriach własnych klas i obiektów systemu (okna, suwaki etc.), a nie klas naszej aplikacji
- Tracimy informacje o typie obiektów przekazywanych do systemu i później przekazywanych nam z powrotem
- Potrzebujemy operacji pozwalającej na odtworzenie "zagubionego" typu obiektu

Operator *dynamic_cast*

- Operator *dynamic_cast* przekazuje poprawny wskaźnik, gdy obiekt ma spodziewany typ, a wskaźnik zerowy w przeciwnym przypadku

```
void my_event_handler(BBwindow* pw)
{
 if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))
                                        // does pw point to an Ival_box?
   pb->do_something() ;
 else {
      // Oops! unexpected event
      }
}
```



- W przypadku wielodziedziczenia, oprócz rzutowania w dół (do klasy pochodnej) i rzutowania w górę (do klasy podstawowej) może występować również rzutowanie skrośne (do klasy siostrzanej)

Operator *dynamic_cast*

- Operator *dynamic_cast* przyjmuje dwa argumenty: typ w nawiasach <> oraz wskaźnik lub referencję w nawiasach ()
- Przy konwersji

dynamic_cast<T>(p)*

jeżeli *p* jest typu *T** lub dostępną klasą podstawową klasy *T*, to wynik jest dokładnie taki sam, jak byśmy po prostu przypisali *p* na *T**, np.:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
 // ...
};
void f(BB_ival_slider* p)
{
 Ival_slider* pi1 = p; // ok
 Ival_slider* pi2 =dynamic_cast<Ival_slider*>(p) ; // ok
 BBslider* pbb1 =p; // error: BBslider is a protected base
 BBslider* pbb2 = dynamic_cast<BBslider*>(p) ; // ok: pbb2 becomes 0
}
```

Operator *dynamic_cast*

- Poprzedni przypadek jest mało interesujący, ale ilustruje fakt, że dynamiczne rzutowanie nie pozwala na przypadkowe naruszenie ochrony prywatnych i chronionych klas podstawowych
- Celem dynamicznego rzutowania jest radzenie sobie wówczas, jeżeli kompilator nie może określić poprawności konwersji. Wtedy operator

  dynamic_cast<T>(p)*

  sprawdza obiekt wskazywany przez *p* (jeżeli taki istnieje). Jeśli ten obiekt pochodzi z klasy *T* lub ma unikatową klasę podstawową typu *T*, to *dynamic_cast* przekazuje wskaźnik typu *T** do tego obiektu, w przeciwnym razie 0.
- Jeżeli *p* ma wartośc zero, to wynikiem operacji jest również zero.

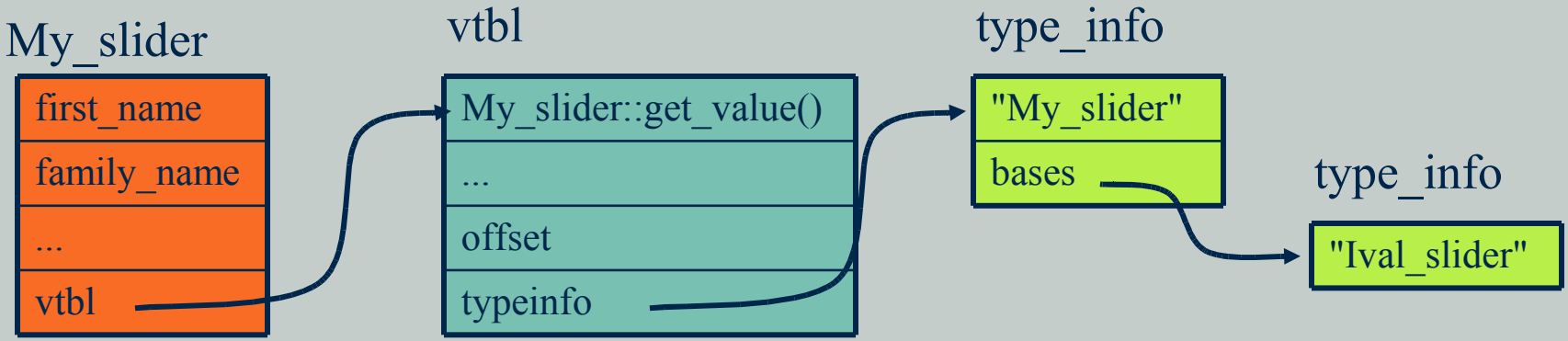
Operator *dynamic_cast*

- Aby wykonać rzutowanie w dół lub skrośne, *dynamic_cast* wymaga wskaźnika lub referencji do typu polimorficznego

```
class My_slider: public Ival_slider { // polymorphic base
                                      //(Ival_slider has virtual functions)
 // ...
};
class My_date : public Date { // base not polymorphic
                              // (Date has no virtual functions)
 // ...
};
void g(Ival_box* pb, Date* pd)
{
  My_slider* pd1 = dynamic_cast<My_slider*>(pb) ; // ok
  My_date*pd2 =dynamic_cast<My_date*>(pd) ; // error: Date not polymorphic
}
```

Operator *dynamic_cast*

- Wymaganie, by typ wskaźnikowy był polimorficzny, upraszcza implementację dynamicznego rzutowania, ponieważ ułatwia znalezienie miejsca na przechowanie niezbędnych informacji o typie obiektu
- Typowa implementacja dołącza do obiektu "obiekt z informacją o typie", umieszczając wskaźnik do informacji o typie w tablicy metod wirtualnych obiektu



- *Offset* jest przesunięciem, pozwalającym na znalezienie początku pełnego obiektu, gdy ma się tylko wskaźnik do polimorficznego podobiektu

Operator *dynamic_cast*

- Docelowy typ dynamicznego rzutowania nie musi być polimorficzny. Pozwala to zapakować typ konkretny w polimorficzny, np. w celu przesłania przez obiektowy system wejścia-wyjścia, a później wypakować typ konkretny:

```
class Io_obj{ // base class for object I/O system
 virtual Io_obj* clone() = 0;
};
class Io_date : public Date, public Io_obj{ };
void f(Io_obj* pio)
{
 Date* pd = dynamic_cast<Date*>(pio) ;
 // ...
}
```

- Można użyć dynamicznego rzutowania do void*, aby określić adres początku obiektu typu polimorficznego, np:

```
void g(Ival_box* pb,Date* pd)
{
 void* pd1 = dynamic_cast<void*>(pb) ; // ok
 void* pd2 =dynamic_cast<void*>(pd) ; // error: Date not polymorphic
}
```

Dynamiczne rzutowanie referencji

- Aby uzyskać polimorficzne zachowanie, trzeba dostawać się do obiektu przez wskaźnik lub referencję.
- Gdy używa się dynamicznego rzutowania do typu wskaźnikowego, to 0 oznacza niepowodzenie.
- W przypadku rzutowania referencji, zgłaszany jest wyjątek *bad_cast*

```
void f(Ival_box* p, Ival_box& r)
{
 if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {
                                          // does p point to an Ival_slider?
   // use 'is'
 } else {
   // *p not a slider
 }
 Ival_slider& is = dynamic_cast<Ival_slider&>(r) ;
                                          // r references an Ival_slider!
   // use 'is'
}
```

- Jeśli użytkownik chce być chroniony przed złym rzutowaniem referencji, musi dostarczyć odpowiednią procedurę obsługi

```
void g()
{
 try {
   f(new BB_ival_slider,*new BB_ival_slider) ;
                                          // arguments passed as Ival_boxs
   f(new BBdial,*new BBdial) ; // arguments passed as Ival_boxs
 }
 catch (bad_cast) {
   // ...
 }
}
```

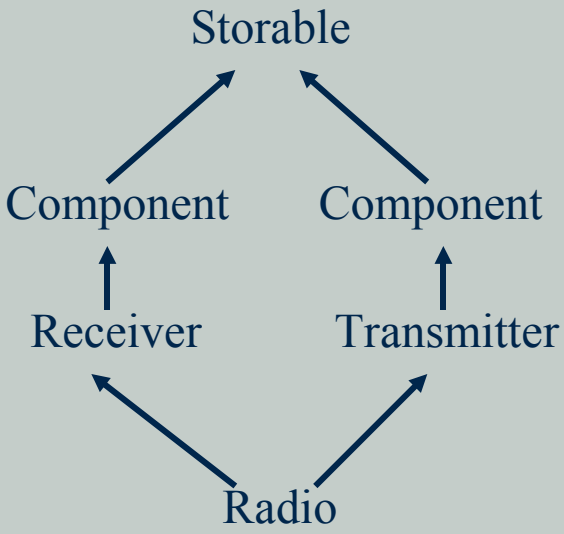
Nawigacja po hierarchii klas

- Gdy używa się pojedynczego dziedziczenia, to klasa i jej klasy pochodne tworzą drzewo zakorzenione w jednej klasie podstawowej
- Gdy używa się wielodziedziczenia, to nie ma jednego korzenia.
- Jeżeli ta sama klasa pojawia się w hierarchii więcej niż jeden raz, to musimy być ostrożni, odnosząc się do obiektu lub obiektów reprezentujących tę klasę

Nawigacja po hierarchii klas

- Rozważmy następującą kratę klas

```
class Component : public virtual Storable
{ /* ... */ };
class Receiver : public Component
{ /* ... */ };
class Transmitter : public Component
{ /* ... */ };
class Radio : public Receiver, public
Transmitter{ /* ... */ };
```



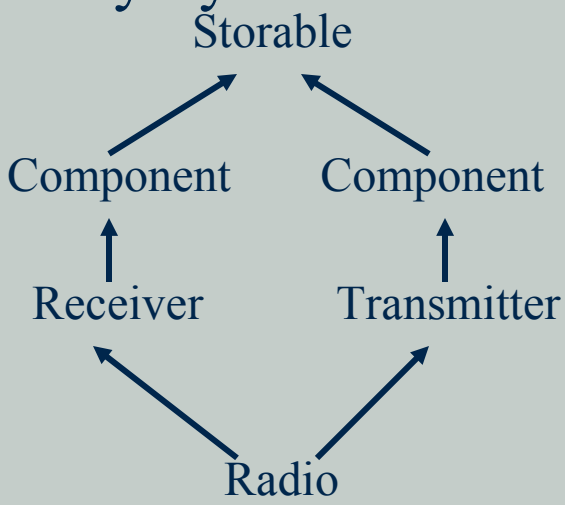
- Obiekt *Radio* ma dwa podobiekty klasy *Component*. W rezultacie dynamiczne rzutowanie z *Storable* do *Component* w *Radio* będzie niejednoznaczne i przekaże 0. Nie ma sposobu na określenie, o który *Component* chodziło programiście

```
void h1(Radio& r)
{
 Storable* ps= &r;
 // ...
 Component* pc = dynamic_cast<Component*>(ps) ; // pc = 0
}
```

Nawigacja po hierarchii klas

- Takiej niejednoznaczności nie można zwykle wykryć w czasie kompilacji

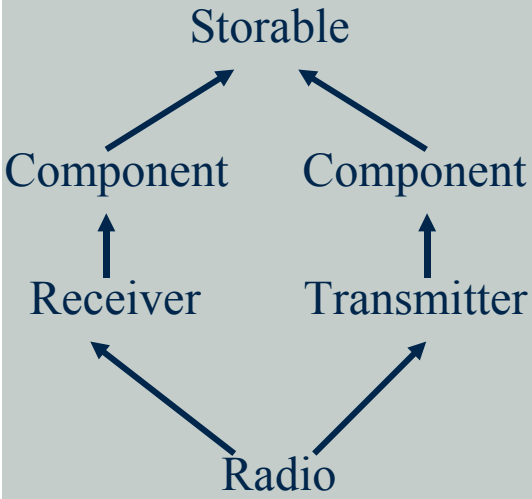
```
void h2(Storable* ps) // ps might or might not
                      // point to a Component
{
 Component* pc = dynamic_cast<Component*>(ps) ;
 // ...
}
```



- Wykrywanie w czasie wykonania niejednoznaczności tego rodzaju jest potrzebne tylko w odniesieniu do wirtualnych klas podstawowych. Zwykłe klasy podstawowe podczas rzutowania w dół (w kierunku klasy pochodnej) zawsze mają unikatowy podobiekt danego rzutowania (lub żaden).
- Równoważna niejednoznaczność pojawia się podczas rzutowania w górę (w kierunku klasy podstawowej). Taką niejednoznaczność można wykryć w czasie kompilacji

Rzutowania statyczne i dynamiczne

- Operator *dynamic_cast* może rzutować z polimorficznej wirtualnej klasy podstawowej do klasy pochodnej lub siostrzanej. Operator *static_cast* nie bada rzutowanego obiektu, więc nie może rzutować




```
void g(Radio& r)
{
 Receiver* prec= &r; // Receiver is ordinary base of Radio
 Radio* pr = static_cast<Radio*>(prec) ; // ok, unchecked
 pr = dynamic_cast<Radio*>(prec) ; // ok, runtime checked
 Storable* ps= &r; // Storable is virtual base of Radio
 pr = static_cast<Radio*>(ps) ;
        // error: cannot cast from virtual base
 pr = dynamic_cast<Radio*>(ps) ; // ok, runtime checked
}
```

- Operator dynamicznego rzutowania wymaga polimorficznego argumentu.
- Z użyciem operatora *dynamic_cast* wiąże się niewielki koszt czasu wykonania. Jeżeli w programie stosuje się inne sposoby zapewnienia, że rzutowanie jest poprawne, można stosować *static_cast*.

Rzutowania statyczne i dynamiczne

- Kompilator nie otrzymuje informacji o pamięci wskazywanej przez *void**. Do rzutowania z *void** jest potrzebny *static_cast*:

```
Radio* f(void* p)
{
 Storable* ps = static_cast<Storable*>(p) ; // trust the programmer
 return dynamic_cast<Radio*>(ps) ;
}
```

- Zarówno *dynamic_cast* jak i *static_cast* respektują *const* i kontrolę dostępu, np:

```
class Users : private set<Person> { /* ... */ };
void f(Users* pu, const Receiver* pcr)
{
 static_cast<set<Person>*>(pu) ; // error: access violation
 dynamic_cast<set<Person>*>(pu) ; // error: access violation
 static_cast<Receiver*>(pcr) ; // error: can't cast away const
 dynamic_cast<Receiver*>(pcr) ; // error: can't cast away const
 Receiver* pr = const_cast<Receiver*>(pcr) ; // ok
 // ...
}
```

- Nie można rzutować do prywatnej klasy podstawowej, a usunięcie *const* rzutowaniem wymaga użycia *const_cast*. Jednak wynik jest bezpieczny tylko wtedy, gdy obiektu nie zadeklarowano pierwotnie jako *const*.

Podsumowanie operatorów rzutowania

- *static_cast*
  - niesprawdzone rzutowanie między typami spokrewnionymi
- *dynamic_cast*
  - sprawdzone rzutowanie między typami spokrewnionymi
- *const_cast*
  - usunięcie atrybutu const z obiektu
- *reinterpret_cast*
  - rzutowanie między typami niespokrewnionymi (np. *int* i wskaźnik)
- Rzutowanie w stylu C *(T)e*
  - dowolna konwersja, jaką można wyrazić jako kombinację operatorów *static_cast*, *reinterpret_cast* i *const_cast*

Konstrukcja i destrukcja obiektu klasy

- Obiekt klasy jest budowany z "surowej pamięci" za pomocą swoich konstruktorów i wraca do stanu "surowej pamięci" po wykonaniu swoich destruktorów
- Konstrukcja przebiega z dołu do góry, destrukcja z góry na dół.
- Jeśli konstruktor klasy *Component* wywoła funkcję wirtualną, to będzie to wersja zdefiniowana dla *Storable* lub *Component*, ale nie ta dla *Receiver*, *Transmitter* lub *Radio*. Na tym etapie konstrukcji obiekt nie jest jeszcze obiektem *Radio*, lecz jedynie częściowo skonstruowanym obiektem.
- Lepiej unikać wywoływania funkcji wirtualnych podczas konstrukcji i destrukcji

Operator *typeid*

- Operator *typeid* zwraca obiekt reprezentujący typ swojego argumentu
- *typeid* zachowuje się jak funkcja o następującej deklaracji:

```
class type_info;
const type_info& typeid(type_name) throw(bad_typeid) ;// pseudo declaration
const type_info& typeid(expression) ; // pseudo declaration
```

- *type_info* jest zdefiniowany w bibliotece standardowej, w pliku nagłówkowym <*typeinfo*>
- Najczęściej *typeid()* używa się do znalezienia typu obiektu wskazanego wskaźnikiem lub referencją:

```
void f(Shape& r, Shape* p)
{
 typeid(r) ; // type of object referred to by r
 typeid(*p) ; // type of object pointed to by p
 typeid(p) ; // type of pointer, that is, Shape*
             // (uncommon, except as a mistake)
}
```

- Jeżeli wartością wskaźnika jest 0, to *typeid()* zgłasza wyjątek *bad_typeid*

Operator *typeid*

- Niezależna od implementacji część *type_info* wygląda następująco:

```
class type_info {
 public:
   virtual ~type_info() ; // is polymorphic
   bool operator==(const type_info&) const; // can be compared
   bool operator!=(const type_info&) const;
   bool before(const type_info&) const; // ordering
   const char* name() const; // name of type
 private:
   type_info(const type_info&) ; // prevent copying
   type_info& operator=(const type_info&) ; // prevent copying
   // ...
};
```

- Metoda *before()* umożliwia sortowanie obiektów. Nie ma związku między zależnościami definiowanymi przez *before()*, a relacjami dziedziczenia
- Nie gwarantuje się istnienia dokładnie jednego obiektu *type_info* dla każdego typu w systemie
  - równość należy testować używając == na obiektach *type_info*, a nie na wskaźnikach do takich obiektów

Operator *typeid*

- Czasami trzeba znać właściwy typ obiektu, by wykonać pewną standardową usługę na całym obiekcie (a nie jedynie na pewnej klasie podstawowej tego obiektu)
- Idealne byłoby, gdyby takie usługi dostępne były jako funkcje wirtualne, by nie trzeba było znać właściwego typu
- Czasami nie można założyć istnienia wspólnego interfejsu dla każdego obsługiwanego obiektu, konieczne więc jest obejście tego problemu przez wykorzystanie znajomości właściwego typu
- Inne zastosowanie to uzyskanie nazwy klasy w celach diagnostycznych:

```
#include<typeinfo>
void g(Component* p)
{
 cout << typeid(*p).name() ;
}
```

- Znakowa reprezentacja nazwy zależy od implementacji.
- Użyty tutaj napis w stylu C jest umieszczony w pamięci zarządzanej prze system, więc programista nie powinien próbować wykonywać na nim *delete []*

Użycie i nadużycie RTTI

- RTTI = Run Time Type Information
- Jawnej informacji o typie w czasie wykonania powinno się używać tylko wtedy, gdy jest to konieczne
- Kontrola statyczna (w czasie kompilacji) jest bezpieczniejsza, generuje mniejszy narzut i umożliwia pisanie programów o lepszej strukturze
- Można użyć RTII do napisania kiepsko zamaskowanej instrukcji *switch*:

```
// misuse of runtime type information:
void rotate(const Shape& r)
{
 if (typeid(r) == typeid(Circle)) {
   // do nothing
 }
 else if (typeid(r) == typeid(Triangle)) {
   // rotate triangle
 }
 else if (typeid(r) == typeid(Square)) {
   // rotate square
 }
// ...
}
```

- W takiej sytuacji lepiej byłoby użyć funkcji wirtualnych

Wskaźniki do składowych

- Wskaźniki do funkcji są przydatne, kiedy klasa ma wiele składowych z takimi samymi argumentami

```
class X {
 double g(double a) { return a*a + 5.0; }
 double h(double a) { return a - 13; }
public:
 void test(X*, X);
};
typedef double (X::*pf)(double);// pointer to member
void X::test(X* p, X q) {
 pf m1 = &X::g;
 pf m2 = &X::h;
 double g6 = (p->*m1)(6.0); // call through pointer to member
 double h6 = (p->*m2)(6.0); // call through pointer to member
 double g12 = (q.*m1)(12); // call through pointer to member
 double h12 = (q.*m2)(12); // call through pointer to member
}
int main(){
 X i;
 i.test(&i, i);
}
```

- ->* i *. są specjalnymi operatorami do obsługi wskaźników do składowych
- Wskaźnik do składowej statycznej jest zwykłym wskaźnikiem

Wskaźniki do składowych

- Funkcje wirtualne działają jak zwykle

```
class X
{
public:
  virtual void f (double a)  {
    cout << "X::f with parameter "<<a<<endl;  }
 virtual ~X(){};
};
class Y: public X
{
public:
  void f (double a)  {
    cout << "Y::f with parameter "<<a<<endl;  }
};
typedef void (X::*pf) (double); // pointer to member
void test (X * p, X * q)
{
  pf m = &X::f;
  (p->*m)(6.0);
  (q->*m)(7.0);
};
int main () {
  X i;   Y j;
  test (&i, &j);
}
```

- Wynika stąd, że wskaźniki do składowych wirtualnych nie są adresami, są przesunięciami w tablicy metod wirtualnych
- Wskaźniki do składowych wirtualnych można wymieniać między przestrzeniami adresowymi

Wskaźniki do składowych i dziedziczenie

- Klasa pochodna ma co najmniej te składowe, które odziedziczyła od swoich klas podstawowych, często ma ich więcej
- Oznacza to, że bezpiecznie możemy przypisać wskaźnik do składowej klasy podstawowej do wskaźnika do składowej klasy pochodnej, ale nie odwrotnie

```
class X {
 public:
   virtual void start() ;
   virtual ~X() {}
};
class Y : public X {
 public:
   void start() ;
   virtual void print() ;
};
void (X::* pmi)() = &Y::print; // error
void (Y::*pmt)() = &X::start; // ok
```

Operator new i delete

- Operatory obsługujące pamięć wolną (*new*, *delete*, *new []* i *delete[]*) są zaimplementowane za pomocą funkcji

```
void* operator new(size_t) ; // space for individual object
void operator delete(void*) ;
void* operator new[](size_t) ; // space for array
void operator delete[](void*) ;
```

- Kiedy operator *new* ma przydzielić pamięć dla obiektu, wywołuje *operator new*(), który przydziela odpowiednią liczbę bajtów. Podobnie, kiedy *new* ma przydzielić pamięć na tablicę, wywołuje *operator new[]()*.
- Kiedy *new* nie będzie mogło znaleźć wolnej pamięci do przydziału, domyślnie zgłoszony zostanie wyjątek *bad_alloc*
- Możemy określić, co ma zrobić *new*, gdy wyczerpie się pamięć. Kiedy *new* kończy się niepowodzeniem, najpierw wywołuje funkcję podaną jako argument wywołania funkcji *set_new_handler()*, zadeklarowanej w <*new*>

```
void out_of_store() {
  cerr << "operator new failed: out of store\n";
  throw bad_alloc() ;
}
int main() {
  set_new_handler(out_of_store) ; // make out_of_store the new_handler
  for (;;) new char[10000] ;
    cout << "done\n";
}
```

Operator new i delete

- Można tak zaprogramować funkcję obsługi, aby można było zrobić coś bardziej inteligentnego, niż przerwanie działania programu
- Jeśli programista wie, jak działają funkcje *new* i *delete* (np. jeżeli dostarczył własny operator *new ()* i operator *delete()*), to może napisać taką funkcję obsługi błędu, za pomocą której można będzie znaleźć dla *new* trochę pamięci
- Operator *new()* zaimplementowany z użyciem funkcji malloc może wyglądać następująco:

```
void* operator new(size_t size)
{
 for (;;) {
   if (void* p =malloc(size)) return p; // try to find memory
   if (_new_handler == 0) throw bad_alloc() ; // no handler: give up
   _new_handler() ; // ask for help
 }
}
```

- Wynika stąd, że funkcja obsługi może zachować się na dwa sposoby:
  - znaleźć więcej pamięci i wrócić
  - zgłosić wyjątek bad_alloc

Umieszczający operator *new*

- Możemy umieścić obiekt pod dowolnym adresem, używając umieszczającego operatora new

```
void* operator new(size_t, void* p) { return p; }
                                          // explicit placement operator

int main()
{
 char buf[sizeof(string)];
 string* s = new(buf) string; // construct an string at 'buf;' invokes:
                              // operator new(sizeof(string),buf);
 *s="hello";
 cout << *s<<endl;
 s->~string();
};
```

- Jest to jeden z nielicznych przypadków, kiedy jawnie wywołuje się destruktor obiektu
- Ta wersja jest najprostszą wersją umieszczającego operatora *new*. Jest zdefiniowana w pliku nagłówkowym *<new>*

Umieszczający operator *new*

- Umieszczający operator *new* można również wykorzystać do przydziału pamięci z określonej strefy:

```
class Arena {
 public:
   virtual void* alloc(size_t) =0;
   virtual void free(void*) =0;
   // ...
};
void* operator new(size_t sz,Arena* a) {
 return a->alloc(sz) ;
}
```

- Obiektom dowolnych typów w miarę potrzeby można przydzielać pamięć z różnych stref

```
extern Arena*Persistent;
extern Arena* Shared;
void g(int i) {
 X* p = new(Persistent)X(i) ; // X in persistent storage
 X* q = new(Shared) X(i) ; // X in shared memory
 // ...
}
```

- Destruktor w dalszym ciągu trzeba wywołać jawnie

```
void destroy(X* p,Arena* a) {
 p->~X() ; // call destructor
 a->free(p) ; // free memory
}
```

Umieszczający operator *delete*

- Umieszczający operator *delete* jest wywoływany w przypadku wystąpienia wyjątku w konstruktorze tworzonego obiektu

```
void operator delete (void *s, Arena * a)
{
  a->free (s);
};
```

- Oprócz skalarnych operatorów umieszczających *new* i *delete* można również zdefiniować podobne operatory dla tablic

Alokacja pamięci dla klas

- Można samemu przejąć zarządzanie pamięcia dla klasy, definiując w niej *operator new()* i *operator delete()*

```
class Employee {
 // ...
 public:
 // ...
   void* operator new(size_t) ;
   void operator delete(void*, size_t) ;
};
```

- Składowe *operator new()* i *operator delete()* są niejawnie składowymi statycznymi

```
void* Employee::operator new(size_t s)
{
 // allocate 's' bytes of memory and return a pointer to it
}
void Employee::operator delete(void* p, size_t s)
{
 // assume 'p' points to 's' bytes of memory
 // allocated by Employee::operator new()
 // and free that memory for reuse
}
```

Alokacja pamięci dla klas

- Dzięki argumentowi typu *size_t* w operatorze *delete*, w funkcji przydziału pamięci można uniknąć zapamiętywania informacji o rozmiarze podczas każdego przydziału
- W przypadku, kiedy obiekt jest zwalniany poprzez wskaźnik do jego klasy bazowej, pojawia się problem z podaniem odpowiedniego rozmiaru operatorowi *delete*

```
class Manager : public Employee {
 int level;
 // ...
};
void f()
{
 Employee* p = new Manager; // trouble (the exact type is lost)
 delete p;
}
```

- W celu uniknięcia problemu trzeba w klasie bazowej umieścić wirtualny destruktor. Może on być nawet pusty.

```
class Employee {
 public:
    void* operator new(size_t) ;
    void operator delete(void*, size_t) ;
    virtual ~Employee() ;
    // ...
};
Employee::~Employee() { }
```

Przydział pamięci na tablicę

- Dla klasy można zdefiniować również tablicowe operatory alokacji i dealokacji pamięci

```
class Employee {
 public:
   void* operator new[](size_t) ;
   void operator delete[](void*, size_t) ;
   // ...
};
void f(int s)
{
 Employee* p = new Employee[s] ;
 // ...
 delete[] p;
}
```

- Pamięci dostarczy wywołanie

$$Employee::operator\ new[](sizeof(Employee)*s+delta)$$

gdzie delta jest pewnym narzutem zależnym od implementacji, a zwolni ją wywołanie

$$Employee::operator\ delete[](p,s*sizeof(Employee)+delta)$$

Obiekty tymczasowe

- Obiekty tymczasowe najczęściej powstają podczas wartościowania wyrażeń arytmetycznych, np. podczas obliczania x*y+z częściowy rezultat x*y musi być gdzieś przechowywany
- Obiekt tymczasowy jest niszczony po zakończeniu obliczania pełnego wyrażenia, w którym został stworzony, chyba, że jest związany z referencją (wtedy później) lub użyto go do zainicjowania nazwanego obiektu (wtedy może być zniszczony wcześniej). Pełne wyrażenie to takie, które nie jest podwyrażeniem żadnego innego

```
void f(string& s1, string& s2, string& s3)
{
 const char* cs= (s1+s2).c_str() ;
 cout << cs;
 if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
      // cs used here
 }
}
```

Wskaźnik do zwolnionego obszaru pamięci

- Do przechowania *s1+s2* tworzy się tymczasowy obiekt klasy *string*. Następnie wyłuskuje się z niego wskaźnik do napisu w stylu C. Wreszcie, usuwa się obiekt tymczasowy.
- Warunek instrukcji warunkowej zadziała zgodnie z oczekiwaniami. Nie ma jednak gwarancji, że użycie *cs* wewnątrz instrukcji warunkowej będzie poprawne

Obiekty tymczasowe

- Można użyć obiektu tymczasowego jako inicjatora dla referencji z atrybutem *const* lub nazwanego obiektu

```
void g(const string&, const string&) ;
void h(string& s1, string& s2)
{
 const string& s = s1+s2;
 string ss = s1+s2;
 g(s,ss) ; // we can use s and ss here
}
```

- Można również utworzyć obiekt tymczasowy, jawnie wywołując konstruktor. Takie obiekty tymczasowe są niszczone dokładnie tak samo, jak obiekty generowanie niejawnie.

```
void f(Shape& s, int x, int y)
{
 s.move(Point(x,y)) ; // construct Point to pass to Shape::move()
 // ...
}
```