

# Programowanie Obiektowe i C++

Marcin Benke

Instytut Informatyki UW

2.10.2006

# Podstawowe elementy proceduralne w C++

- Program i wyjście
- Zmienne i arytmetyka
- Wskaźniki i tablice
- Testy i pętle
- Funkcje

# Pierwszy program

```
// Niezbędne zaklęcia przygotowawcze ;-)  
#include <iostream>  
using namespace std ;  
  
main() // Program główny  
{  
    // do obiektu cout wysyłamy komunikat <<  
    // argument - co wypisać  
    cout << "Hello, brave new world" ;  
    cout << endl ; // koniec linii  
}
```

# Zmienne i arytmetyka

- Każda nazwa i każde wyrażenie mają typ, np

```
int i ;
```

deklaruje zmienną `i` typu `int`, tj. całkowitą.

- Typami podstawowymi są:

```
char
```

```
short
```

```
int
```

```
long
```

reprezentujące liczby całkowite oraz

```
float
```

```
double
```

```
long double
```

reprezentujące liczby zmiennopozycyjne.

# Operacje arytmetyczne

- Operatory arytmetyczne można stosować do dowolnej kombinacji typów podstawowych

+ (plus, jedno- i dwuaargumentowy)

- (minus, jedno- i dwuaargumentowy)

\* (mnożenie)

/ (dzielenie)

% (reszta z dzielenia)

# Relacje

- To samo tyczy się operatorów relacji:

`==` (równe)

`!=` (nierówne)

`<` (mniejsze)

`>` (większe)

`<=` (mniejsze lub równe)

`>=` (większe lub równe)

- Wynik porównania jest typu `int`.
- **Pamiętaj:** `symbol =` oznacza przypisanie, a nie porównanie!
- Wyrażenie zakończone średnikiem staje się instrukcją.

# Wskaźniki i tablice

- Tablicę można zadeklarować następująco:

```
char t[10]; // tablica 10 znaków
```

- podobnie można zadeklarować wskaźnik do znaku

```
char* p; // wskaźnik do znaku
```

- Dolnym zakresem wszystkich tablic jest zero.

- Tablica `t` ma elementy `t[0]..t[9]`

- Wskaźnik przechowuje adres obiektu

```
p = &t[3]; // p wskazuje czwarty element
```

- Operator jednoargumentowy `&` daje w wyniku adres argumentu.

# Bloki

```
{ double d;  
  int i;  
  short s;  
  
  d = d+i;  
  i = s*i;  
}
```

- Każdą instrukcję kończymy średnikiem
- Ciąg instrukcji otoczony nawiasami klamrowymi jest instrukcją
- Deklaracje w bloku są widoczne tylko wewnątrz bloku



# Testy

```
if (i % 2) {  
    i = i / 2 ;  
} else  
    i = 3 * i + 1 ;
```

- Instrukcja warunkowa:

`if (warunek) instrukcja else instrukcja`

- Pierwsza instrukcja zostanie wykonana jeśli *warunek* ma wartość różną od zera.
- Forma skrócona (bez `else`):  
`if (warunek) instrukcja`

# Pętla while

```
int i ;
cin >> i ; // pobierz i z wejscia
while (i != 0) { // albo: while(i)
    if (i % 2)
        i = i / 2 ;
    else
        i = 3 * i + 1 ;
}
```

- Ciało pętli będzie wykonywane tak długo jak warunek będzie miał wartość niezerową.

# Pętla for

## Konstrukcja

```
for(wyrażenie1; wyrażenie2; wyrażenie3)  
  instrukcja
```

odpowiada konstrukcji

```
wyrażenie1;  
while(wyrażenie2) {  
  instrukcja  
  wyrażenie3  
}
```

# Pętla `for` — przykład

```
for(i = 0; i < 10 ; i++)  
    cout << t[i] ;
```

# Instrukcja do-while

- Składnia:

do *instrukcja* (*wyrażenie*)

- Przykład

```
do {  
    char ch ;  
    cin >> ch ;  
    cout << ch ;  
} while(ch != 'q') ;
```

- Pętla wykonuje się tak długo jak warunek jest prawdziwy.
- Warunek jest sprawdzany na końcu pętli.

# Instrukcje `break` i `continue`

- Instrukcja `break` wewnątrz pętli powoduje natychmiastowe wyjście z niej.
- W przypadku zagnieżdżonych pętli wyjście dotyczy tylko jednej (wewnętrznej) pętli.
- Instrukcja `continue` powoduje natychmiastowe przejście do sprawdzania warunku pętli.

# Przykład break i continue

```
for(;;) { // pętla nieskończona
    char ch ;
    cin >> ch ;
    if(ch == 'q') break ; // Wyjdź z pętli
    if(ch == 'a') {
        cout << 'Ala ma kota << endl ;
        continue ; // Następny obrót pętli

    if(ch == 'b') cout << "Beeee!" << endl ;
}
}
```

# Instrukcja `switch`

- Instrukcja `switch` pozwala na wybór wariantu w zależności od wartości podanego wyrażenia całkowitego.
- Składnia:

```
switch(wyrażenie) {  
  case stała1:  
    ciąg instrukcji  
    // ...  
  case stałan:  
    ciąg instrukcji  
  default:  
    ciąg instrukcji  
}
```



# Instrukcja `switch` c.d.

- Każdy z ciągów instrukcji powinien w zasadzie kończyć się instrukcją `break`, w przeciwnym wypadku zostanie wykonany również następny ciąg instrukcji...
- Ciąg instrukcji z etykietą `default` zostanie wykonany jeśli wartość wyrażenia nie pasuje do żadnego z wymienionych przypadków.

# Przykład switch

```
int quit = 0 ;
do {
    char ch ;
    cin >> ch ;
    switch(ch) {
    case 'q' :
        quit = 1 ;
        break ; // Wyjdź ze switch
    case 'a' :
        cout << 'Ala ma kota << endl ;
        break ;
    // ...
    }
} while(!quit)
```

# Deklaracje funkcji

- Deklaracja funkcji umożliwia użycie funkcji zdefiniowanej później lub w innym pliku.
- Deklaracja funkcji podaje typ wyniku funkcji, jej nazwę oraz typy argumentów.
- Można (ale nie trzeba) podać nazwy argumentów.
- Typ wyniku *void* oznacza, że funkcja nie daje wyniku
- Deklarację funkcji kończymy średnikiem.
- Przykłady:

```
void drawLine(int x1, int x2, int y1, int y2) ;  
int foo(int, float) ;
```

# Definicje funkcji

- Definicja funkcji podaje jej nagłówek (podobnie jak deklaracja, z tym, że tu nazwy argumentów są obowiązkowe) oraz treść
- Treść funkcji jest zawsze blokiem
- Definicji funkcji **nie** kończymy średnikiem.
- Przykład:

```
void paint ()  
{  
    drawLine (100, 100, 300, 100) ;  
}
```

# Powrót z funkcji i dawanie wyniku

- Instrukcja `return` powoduje natychmiastowy powrót z funkcji
- Jeśli funkcja daje wynik (typ wyniku różny od `void`), argumentem dla `return` jest wyrażenie, którego wartość stanie się wynikiem funkcji.

- Przykład:

```
char cfunc(int i) {  
    switch(i) {  
        case 0: return 'a';  
        case 1: return 'g';  
    } //switch  
} //cfunc
```

- Zwróćmy uwagę, że po `return` użycie `break` jest zbędne.

# Rekurencja

- Podobnie jak w innych językach możemy definiować funkcje rekurencyjne.
- Każde wcielenie funkcji ma własne wartości parametrów i zmiennych lokalnych.

```
int silnia(int n) {  
    if(n<=1)  
        return 1;  
  
    int s ;  
    s = silnia(n-1);  
    return n * s ;  
}
```

# Modyfikacja obiektów zewnętrznych

- Argumenty funkcji są przekazywane “przez wartość”, tj. funkcja otrzymuje kopię wartości argumentu.
- Zmiana wartości argumentu wewnątrz funkcji jest widoczna tylko wewnątrz funkcji

```
void f(int a) {  
    a = 5;  
}  
  
int main {  
    int x = 42 ;  
    f(x) ; // x ma nadal wartość 42  
}
```

- Jeśli chcemy zmienić wartość obiektu “zewnętrznego” (z punktu widzenia funkcji) możemy przekazać wskaźnik do niego

# Modyfikacja obiektów zewnętrznych

- Jeśli chcemy zmienić wartość obiektu “zewnętrznego” (z punktu widzenia funkcji) możemy przekazać wskaźnik do niego

```
void g(int* a) {
    *a = 5;
}
int main {
    int x = 42 ;
    g(&x) ; // x ma wartość 5
}
```

- Funkcja `g` otrzymuje adres zmiennej `x` i może operować na niej samej, a nie tylko na jej kopii.



# Referencje

- Język C++ (w odróżnieniu od C) umożliwia również przekazywanie argumentów przez referencję

```
void h(int& a) {  
    a = 5;  
}  
  
int main {  
    int x = 42 ;  
    h(x) ; // x ma wartość 5  
}
```

- Wywołanie funkcji `h` wygląda tak samo jak wywołanie przez wartość, ale w istocie przekazywany jest adres zmiennej `x`, a nie kopia jej wartości.

# Wskaźniki do funkcji

```
void run(void (*paint)()) {  
    //...  
    (*paint)()  
}  
void paint() {  
    drawLine(100,100,300,100) ;  
}  
int main() {  
    run(paint) ;  
}
```

- Argumentem funkcji run jest wskaźnik do funkcji bezwynikowej.

# Wskaźniki do funkcji

```
void run(void (*paint)()) {  
    //...  
    (*paint)()  
}  
void paint() {  
    drawLine(100,100,300,100) ;  
}  
int main() {  
    run(paint) ;  
}
```

- Dla ustalenia adresu funkcji nie używamy operatora & a tylko jej nazwy.

# Wskaźniki do funkcji

```
void run(void (*paint)()) {  
    //...  
    (*paint)()  
}  
void paint() {  
    drawLine(100,100,300,100) ;  
}  
int main() {  
    run(paint) ;  
}
```

- Jeśli `pf` jest wskaźnikiem do funkcji, to do jej wywołania używamy konstrukcji `(*pf)(argumenty)`

# Program w wielu plikach

- Program może być zapisany w wielu plikach źródłowych.
- W C++, podobnie jak w C nie ma jawnego systemu modułów.
- Namiastkę modułu tworzymy zapisując jego interfejs w jednym pliku (często z rozszerzeniem `.h`) a implementację w innym (innych).
- Korzystając z modułu włączamy jego interfejs dyrektywą preprocesora `#include`
- `#include "foo"` działa tak jakby w jej miejscu znalazła się treść pliku `foo`.
- `#include <foo>` oznacza, że pliku `foo` należy szukać w “standardowych miejscach dla nagłówek” (np. `#include <iostream>`).

# Kompilacja i łączenie

- Każdy plik z implementacją kompilujemy do pliku obiektowego z rozszerzeniem `.o` (pod Windows `.obj`), np.

```
g++ -c dtest.cpp
```

```
g++ -c drawing.cpp
```

- Na zakończenie łączymy uzyskane pliki obiektowe w plik wykonywalny, np:

```
g++ -o dtest dtest.o drawing.o
```

- Gotowe biblioteki możemy dołączyć używając opcji `-l`, np

```
g++ -o dtest -lX11 dtest.o drawing.o
```

dołącza bibliotekę zawierającą podstawowe funkcje systemu Xwindow.

# Makefile

Zwykle wygodnie jest użyć programu `make`, opisując proces kompilacji w `Makefile`:

```
LIBS=-lX11
```

```
LDFLAGS=-L/usr/X11R6/lib -g
```

```
CFLAGS=-g
```

```
check: dtest
```

```
./dtest
```

```
drawing.o: drawing.cpp drawing.h
```

```
g++ $(CFLAGS) -c $<
```

```
btest.o: btest.cpp drawing.h
```

```
g++ $(CFLAGS) -c $<
```

```
dtest: dtest.o drawing.o
```

# Makefile

Plik `Makefile` składa się z definicji zmiennych (`CFLAGS`, `LDFLAGS`) oraz reguł.

Reguły mają składnię następującą:

`<cel> : <zależności>`

`<TAB> <polecenia>`

`make` kompiluje tylko te pliki które są nieaktualne, tj. starsze od plików od których zależą.

Korzysta w tym celu z reguł wyspecyfikowanych w pliku , oraz z tzw. reguł domyślnych.